Improving the Competitive Ratio of the Online OVSF Code Assignment Problem

AUTHOR(S):

Miyazaki, Shuichi; Okamoto, Kazuya

Improving the Competitive Ratio of the Online OVSF Code Assignment Problem*

Shuichi Miyazaki[1] and Kazuya Okamoto[2]

[1] Academic Center for Computing and Media Studies, Kyoto University
shuichi@media.kyoto-u.ac.jp
[2] Graduate School of Informatics, Kyoto University okia@kuis.kyoto-u.ac.jp

Abstract. Online OVSF code assignment has an important application to wireless communications. Recently, this problem was formally modeled as an online problem, and performances of online algorithms have been analyzed by the competitive analysis. The previous best upper and lower bounds on the competitive ratio were 10 and 5/3, respectively. In this paper, we improve them to 7 and 2, respectively. We also show that our analysis for the upper bound is tight by giving an input sequence for which the competitive ratio of our algorithm is $7-\varepsilon$ for arbitrary $\varepsilon > 0$.

1 Introduction

Universal Mobile Telecommunication System (UMTS) is one of the third generation (3G) technologies, which is a mobile communication standard. UMTS uses a high-speed transmission protocol Wideband Code Division Multiple Access (W-CDMA) as the primary mobile air interface. W-CDMA was implemented based on Direct Sequence CDMA (DS-CDMA), which allows several users to communicate simultaneously over a single communication channel. DS-CDMA utilizes Orthogonal Variable Spreading Factor (OVSF) code to separate communications [1].

OVSF code is based on an *OVSF code tree*, which is a complete binary tree of height h. The leaves of the OVSF code tree are of level 0 and parents of vertices of level ℓ ($\ell = 0, \ldots, h-1$) are of level $\ell+1$. Therefore the level of the root is h. Fig. 1 shows an OVSF code tree of height 4. Each vertex of level ℓ corresponds to a code of level ℓ. In DS-CDMA, each communication uses a code of the specific level. To avoid interference, we need to assign codes (vertices of an OVSF code tree) to communications so that they are *mutually orthogonal*, namely, in any path from the root to a leaf of an OVSF code tree, there is at most one assigned vertex. However, it is not so easy to serve requests *efficiently* as we will see later.

Erlebach et al. [4] first modeled this problem as an online problem, called the online OVSF code assignment problem, and verified the efficiency of algorithms using the competitive analysis: An input σ consists of a sequence of *a-requests (assignment requests)* and *r-requests (release requests)*. An a-request a_i specifies a required level, denoted $\ell(a_i)$. Upon receiving an a-request a_i, the task of an

* This work was supported by KAKENHI 17700015, $19 \cdot 4017$, and 20300028.

online algorithm is to assign a_i to one of the vertices of level $\ell(a_i)$ of OVSF code tree, so that the orthogonality condition is not broken. It may also reassign other requests (already existing in the tree). An r-request r_i specifies an a-request, denoted $f(r_i)$, which was previously assigned and is still assigned to the current OVSF code tree. When an r-request r_i arrives, the task of an online algorithm is to merely remove $f(r_i)$ from the tree (and similarly, it may reassign other requests in the tree). Each assignment and reassignment causes a cost of 1, but removing a request causes no cost. Without loss of generality, we may assume that σ does not include an a-request that cannot be assigned by any reassignment of the existing requests (in other words, the total bandwidth of requests at any point never exceeds the capacity).

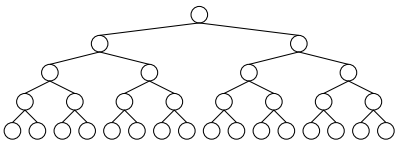

Fig. 1. An OVSF code tree of hight 4.

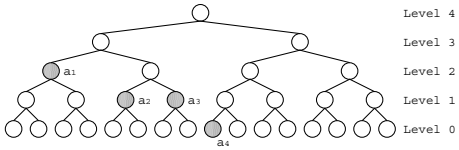

Fig. 2. An example of an assignment.

For example, consider the OVSF code tree given in Fig. 1 and the input $\sigma = (a_1, a_2, a_3, a_4, r_1, a_5)$, where $\ell(a_1) = 2, \ell(a_2) = \ell(a_3) = 1, \ell(a_4) = 0, \ell(a_5) = 3$ and $f(r_1) = a_2$. Suppose that, for the first four requests, an online algorithm assigns a_1 through a_4 as depicted in Fig. 2. Then, r_1 arrives and a_2 is released. Next, a_5 arrives, but an algorithm cannot assign a_5 unless it reassigns other a-requests. If it reassigns a_4 to a child of the vertex to which a_2 was assigned, it can assign a_5 to the right vertex of level 3. In this case, it costs 6 (5 assignments and 1 reassignment). However, if it has assigned a_2 to a vertex in the right subtree of the root, and a_4 to a vertex in the left subtree, then the cost is 5, which is clearly optimal.

This problem also has an application in assigning subnets to users in computer network managements. An IP address space can be divided into subnets, each of which is a fragment of the whole IP address space consisting of a set of continuous IP addresses of size power of 2. This structure can be represented as a complete binary tree, in exactly the same way as our problem. Usually, the sizes of subnets requested by users depend on the number of computers they want to connect to the subnet, and the task of managers is to assign subnets to users so that no two assigned subnets overlap. Apparently, we want to minimize the number of reassignments because a reassignment causes a large cost for updating configurations of computers.

Online algorithms are evaluated by the competitive analysis. The *competitive ratio* of an online algorithm ALG is defined as $\max\{\frac{C_{ALG}(\sigma)}{C_{OPT}(\sigma)}\}$, where $C_{ALG}(\sigma)$ and $C_{OPT}(\sigma)$ are the costs of ALG and an optimal offline algorithm, respectively, for an input σ, and max is taken over all σ. If the competitive ratio of ALG is at most c, we say that ALG is c-competitive.

For the online OVSF code assignment problem, Erlebach et al. [4] developed a $\Theta(h)$-competitive algorithm (recall that h is the hight of the OVSF code tree), and proved that the lower bound on the competitive ratio of the problem is

1.5. Forišek et al. [6] developed a $\Theta(1)$-competitive algorithm, but they did not obtain a concrete constant. Later, Chin, Ting, and Zhang [2] proposed algorithm LAZY by modifying the algorithm of Erlebach et al. [4], and proved that the competitive ratio of LAZY is at most 10. Chin, Ting, and Zhang [2] also showed that no online algorithm can be better than 5/3-competitive.

Our Contribution. In this paper, we improve both upper and lower bounds, namely, we give a 7-competitive algorithm Extended-Lazy, and show that no online algorithm can be better than 2-competitive. We further show that our upper bound analysis is tight by giving a sequence of requests for which the competitive ratio of Extended-Lazy is $7 - \varepsilon$ for an arbitrary constant $\varepsilon > 0$.

Let us briefly explain an idea of improving an upper bound. Erlebach et al. [4] defined the "compactness" of the assignment, and their algorithm keeps compactness at any time. They proved that serving a request, namely assigning (or releasing) a request and modifying the tree to make compact, will cause at most one reassignment at each level, which leads to $\Theta(h)$-competitiveness. Chin, Ting, and Zhang [2] pointed out that always keeping the tree compact is too costly. Their algorithm LAZY does not always keep the compactness but makes the tree compact when it is necessary. To achieve this relaxation, they defined "tanks". By exploiting the idea of tanks, they proved that the cost of serving each request is at most 5, which provides 10-competitiveness. Our algorithm follows this line. We further relax the compactness by defining "semi-compactness". We also use amortized cost analysis. We prove that serving one a-request (r-request, respectively) and keeping semi-compactness costs at most 4 (3, respectively) and obtained 7-competitiveness of Extended-Lazy.

Related Results. For the online OVSF code assignment problem, there are a couple of resource augmentations, namely, online algorithms are allowed to use more bandwidth than an optimal offline algorithm: Erlebach et al. [4] developed a 4-competitive algorithm in which an online algorithm can use a double-sized OVSF code tree. Chin, Zhang, and Zhu [3] developed a 5-competitive algorithm that uses 1/8 extra bandwidth.

Also, there are several *offline* models. One is the problem of finding a minimum number of reassignments to modify the current assignment so that the new request can be assigned, given a current assignment configuration of the tree and a new request. For this problem, Minn and Siu [7] developed a greedy algorithm. Moreover, Erlebach et al. [4] proved that this problem is NP-hard and developed a $\Theta(h)$-approximation algorithm. Another example is an offline version of our problem, namely, given a whole sequence of requests, we are asked to find a sequence of operations that minimizes the number of reassignments. Erlebach, Jacob, and Tomamichel [5] proved that this is NP-hard and gave an exponential-time algorithm.

2 Preliminaries

In this section, we define terminologies needed to give our algorithm, most of which are taken from [2]. Given an assignment configuration, we say that vertex v is *dead* if v or one of its descendants is assigned. In the example of Fig. 3,

shaded vertices are assigned, and vertices with stars (∗) are dead. If, at any level, all the left vertices (on the same level) to the rightmost dead vertex are dead, and all the assigned vertices are mutually orthogonal, then the assignment is called *compact*. For example, the assignment in Fig. 3 is compact.

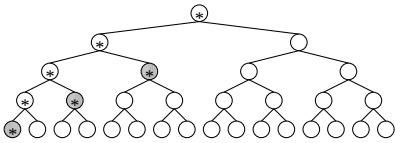

Fig. 3. A compact assignment.

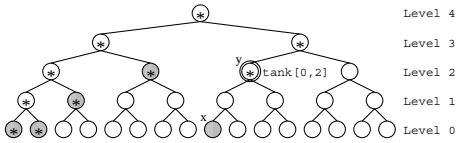

Fig. 4. A semi-compact assignment.

Next, let us define a status of levels. Level ℓ is said to be *rich* if either (i) there is no dead vertices at ℓ or (ii) an a-request of level ℓ can be assigned to the vertex v immediately right of the rightmost dead vertex at ℓ without reassigning other requests. Note that in the case of (ii), none of descendants, ancestors, and v itself is assigned. Otherwise, the level ℓ is said to be *poor*. For example, in the assignment of Fig. 3, levels 0, 2, and 3 are rich and levels 1 and 4 are poor. A level ℓ is said to be *locally rich* if the rightmost assigned vertex is the left child of its parent. For example, in Fig. 3, only level 0 is locally rich. Note that in a compact assignment, locally rich levels are always rich.

Then, we define a *tank*. Consider an a-request a of level b, and suppose that it is assigned to a vertex x of level b. We sometimes consider as if a were assigned to a vertex y of a higher level t ($t > b$) if x is the only assigned descendant of y (see Fig. 4). In this case, the vertex y is called tank$[b, t]$. Levels b and t are called the *bottom* and the *top* of tank$[b, t]$, respectively. We say that level ℓ ($b \leq \ell \leq t$) *belongs to* tank$[b, t]$. Note that we consider that the vertex y is assigned and the vertex x is unassigned.

Finally, let us define the *semi-compactness*. An assignment is said to be *semi-compact* if the following five conditions are satisfied: (i) All the assigned vertices are mutually orthogonal; (ii) All left vertices of the rightmost dead vertex are dead at each level; (iii) Each level belongs to at most 1 tank; (iv) Suppose that there is a tank v(=tank$[b, t]$) at level t. Then level t contains at least one assigned vertex other than v, and there is no dead vertex to the right of v in t; (v) Levels belonging to tanks are all poor except for the top levels of tanks. Fig. 4 shows an example of a semi-compact assignment.

3 Algorithm Extended-Lazy

To give a complete description of Extended-Lazy, we first define the following four functions [2]. Note that a single application of each function keeps the orthogonality, but may break the semi-compactness. However, Extended-Lazy combines functions so that the combination keeps the semi-compactness.

– AppendRich(ℓ, a): This function is available if the level of a-request a is less than or equal to ℓ (namely $\ell(a) \leq \ell$), and level ℓ is rich. It assigns a to the vertex immediately right of the rightmost dead vertex at ℓ. If there is no

dead vertices at ℓ, it assigns a to the leftmost vertex v at ℓ. Note that if $\ell(a) \neq \ell$, this function creates tank$[\ell(a),\ell]$.
- AppendPoor(ℓ, a): This function is available if $\ell(a) \leq \ell$ and level ℓ is poor. It assigns a-request a to the vertex v immediately right of the rightmost dead vertex at ℓ. If there is no dead vertices at ℓ, it assigns a to the leftmost vertex v at ℓ. (If there is no such v, abort.) Note that if $\ell(a) \neq \ell$, tank$[\ell(a),\ell]$ is created. Then, it releases an a-request assigned to a vertex in the path from v to the root and returns it. (Such a request exists because ℓ was poor and v was non-dead. This request is unique because of the orthogonality.)
- FreeTail(ℓ): Release the a-request assigned to the rightmost assigned vertex at level ℓ, and return it.
- AppendLeft(ℓ, a): This function is available if $\ell(a) = \ell$. Assign a-request a to the leftmost non-assigned vertex at level ℓ that has no assigned ancestors or descendants.

Each of AppendRich, AppendPoor, and AppendLeft yields a cost of 1, and FreeTail does not yield a cost.

Now, we are ready to describe Extended-Lazy. Its behaviors on an a-request and an r-request are given in Sects. 3.1 and 3.2, respectively. Executions of Extended-Lazy is divided into several cases. In the description of each case, we explain the behavior of Extended-Lazy, and in addition, for the later analysis, we will calculate the cost incurred and an upper bound on the increase in the number of locally rich levels due to the operations.

3.1 Executions of Extended-Lazy for a-requests

As summarized in Fig. 5, the behavior of Extended-Lazy for an a-request a is divided into six cases based on the status of the level $\ell(a)$ (recall that $\ell(a)$ is the level to which a has to be assigned).

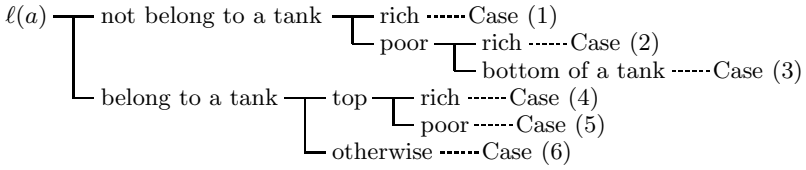

Fig. 5. Execution of Extended-Lazy for an a-request a.

Case (1): The case that $\ell(a)$ does not belong to a tank and is rich. Execute AppendRich$(\ell(a), a)$. The execution of this case costs 1, and the number of locally rich levels increases by at most 1 because only $\ell(a)$ changes status.

Case (2): The case that $\ell(a)$ does not belong to a tank and $\ell(a)$ is poor. Furthermore, if we look at the higher levels from $\ell(a)$ to the root, namely in the order of $\ell(a)+1, \ell(a)+2, \ldots, h$ until we encounter a level that is rich or a bottom of a tank, we encounter a rich level (say, the level t) before a bottom of a tank. In this case, execute AppendRich(t, a). Note that the new tank$[\ell(a), t]$ is created. This case costs 1 and the number of locally rich levels increases by at most 1 since only level t changes status.

Case (3): The same as Case (2), but when looking at higher levels, we encounter a bottom b of tank$[b,t]$ before we encounter a rich level. This case is a little bit complicated. First, execute FreeTail(t) and receive the a-request a' of level b (because a level-b request was assigned to level t by exploiting a tank). Then execute AppendPoor(b,a') (note that b is poor by the condition (v) of semi-compactness), and receive another a-request a'' of level s. Note that there is an assigned vertex at t because of the condition (iv), and hence $b < s \leq t$. Next, using AppendRich(t,a''), assign a'' to the vertex which was tank$[b,t]$, which creates the new tank$[s,t]$ if $s \neq t$. Now, recall that the level b was poor, and hence a' was assigned to a left child. So, the level b is currently locally rich. We execute AppendRich(b,a). Note that tank$[\ell(a),b]$ is newly created. In this case, the cost incurred is 3, and the number of locally rich levels does not change.

Case (4): The case that $\ell(a)$ is the top of a tank and is rich. First, execute FreeTail$(\ell(a))$ and receive the a-request a' from the bottom of the tank. Then, execute AppendRich$(\ell(a),a)$ and AppendRich$(\ell(a),a')$ in this order. Intuitively speaking, we shift the top of the tank to the right, and assign a to the vertex which was a tank. In this case, it costs 2 and similarly as Case (1), the number of locally rich levels increases by at most 1.

Case (5): The case that $\ell(a)$ is the top of a tank, say tank$[b,\ell(a)]$, and is poor. Execute FreeTail$(\ell(a))$ and receive the a-request a' of level b from tank$[b,\ell(a)]$, and execute AppendRich$(\ell(a),a)$ to process a. Note that $\ell(a')$ $(=b)$ is poor because $\ell(a')$ was the bottom of tank$[b,\ell(a)]$. Also, note that b currently does not belong to a tank. Hence our current task is to process a' of level b where b does not belong to a tank and is poor. So, according to the status of levels higher than b, we execute one of Cases (2) or (3). Before going to Cases (2) or (3), the cost incurred is 1 and there is no change in the number of locally rich levels. Hence, the total cost of the whole execution of this case can be obtained by adding one to the cost incurred by the subsequently executed case (Cases (2) or (3)), and the change in the locally rich levels is the same as that of the subsequently executed case.

Case (6): The case that $\ell(a)$ belongs to tank$[b,t]$ and is not the top of tank$[b,t]$. Execute FreeTail(t) and receive the a-request a' of level b from tank$[b,t]$. Then, execute AppendPoor$(\ell(a),a)$ and receive an a-request a'' of level s. By a similar observation as Case (3), we can see that $\ell(a) < s \leq t$. Then, assign a'' to the vertex which was tank$[b,t]$ using AppendRich(t,a''). (Note that tank$[s,t]$ is created if $s \neq t$.) Since $\ell(a)$ was poor, a was assigned to a left child. Hence $\ell(a)$ is currently locally rich. Execute AppendRich$(\ell(a),a')$, which creates tank$[b,\ell(a)]$ if $\ell(a) \neq b$. The execution of this case costs 3, and the number of locally rich levels does not change.

Here we give one remark. Suppose that $\ell(a)$ does not belong to a tank and is poor. Then, we look at higher levels to see which of Cases (2) and (3) is executed. It may happen that we encounter neither a rich level nor a bottom of a tank, namely there is no tank in upper levels, and all upper levels are poor. However, we can claim that this does not happen since such a case happens only when the total bandwidth of all a-requests exceeds the capacity. As we have mentioned previously, we excluded this case from inputs. (Actually, we do not have to exclude this case because our algorithm detects this situation, and in

such a case, we may simply reject the a-request.) Because of the space restriction, we omit the proof of this claim.

The following lemma proves the correctness of Extended-Lazy on a-requests.

Lemma 1. Extended-Lazy *preserves the semi-compactness on a-requests.*

Proof. In order to prove this lemma, we have to show that the five conditions (i) through (v) of semi-compactness are preserved after serving an a-request. It is relatively easy to show that (i) is preserved because Extended-Lazy uses only AppendRich, AppendPoor, and FreeTail, each of which preserves the orthogonality even by a single application. So, let us show that conditions (ii) through (v) are preserved after the execution of each of Cases (1) through (6), provided that (i) through (v) are satisfied before the execution. Because of the space restriction, however, we treat here only Case (1) and omit all other cases.
Case (1): Let v be the vertex (of level $\ell(a)$) to which the request a is assigned. Note that by AppendRich$(\ell(a), a)$, some of ancestors of v may turn from non-dead to dead. It is easy to see that the condition (ii) is preserved at level $\ell(a)$ since we only appended the request to the right of the rightmost dead vertex. Now, suppose that vertex v_s of level s ($s > \ell(a)$) turned from non-dead to dead. Then, by the above observation, v_s is an ancestor of v. Next, let v' be the vertex which is immediately left of v. Then, since v' was dead, v_s is not an ancestor of v'. As a result, the ancestor of v' at level s is the vertex, say v'_s, immediately left of v_s, which implies that v'_s was dead. Thus, condition (ii) is satisfied at any level. It is not hard to see that other conditions, (iii) through (v), are also preserved because Case (1) does not create or remove tanks. □

3.2 Executions of Extended-Lazy for r-requests

Next, we describe executions of Extended-Lazy for r-requests. Similarly as Sec. 3.1, there are eight cases depending on the status of level $\ell(f(r))$ as summarized in Fig. 6, each of which will be explained in the following. Recall that $f(r)$ is the request that r asks to release.

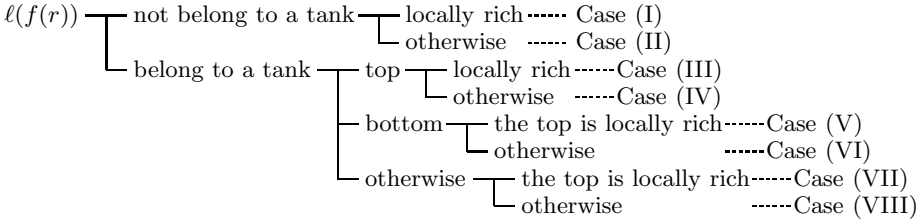

Fig. 6. Execution of Extended-Lazy for an r-request r.

Case (I): The case that $\ell(f(r))$ does not belong to a tank and is locally rich. Release $f(r)$. If $f(r)$ is the rightmost dead vertex at $\ell(f(r))$, do nothing. Otherwise, use FreeTail$(\ell(f(r)))$ and receive an a-request a of level $\ell(f(r))$. Then, using AppendLeft$(\ell(f(r)), a)$, assign a to the vertex to which $f(r)$ was assigned. Note that the vertex v which was the rightmost dead vertex of level $\ell(f(r))$ becomes non-dead after the above operations, which may turn some vertices in

the path from v to the root non-dead from dead. As a result, an assignment may become non-semi-compact. If the semi-compactness is broken, we use the operation Repair, which will be explained later, to retrieve the semi-compactness. The cost of this case is either 1 or 0, and the number of locally rich levels decreases by 1 without considering the effect of Repair. (We later estimate these quantities considering the effect of Repair.)

Case (II): The case that $\ell(f(r))$ does not belong to a tank and is not locally rich. Extended-Lazy behaves exactly the same way as Case (I). Note that vertex v which was the rightmost dead vertex at level $\ell(f(r))$ becomes non-dead after the above operations, but v is a right child because $\ell(f(r))$ was not locally rich. Since the semi-compactness was satisfied before the execution, the vertex immediately left of v was (and is) dead, which implies that the parent and hence all ancestors of v are still dead. Thus, we do not need Repair in this case. It costs either 1 or 0, and the number of locally rich levels increases by 1 because $\ell(f(r))$ becomes locally rich.

Case (III): The case that $\ell(f(r))$ belongs to tank$[b,t]$, $\ell(f(r)) = t$, and t is locally rich. First, release $f(r)$. If $f(r)$ was assigned to the vertex immediately left of tank$[b,t]$ at t, do nothing. Otherwise, using FreeTail(t), receive an a-request a' of level t, and using AppendLeft(t,a'), assign a' to the vertex to which $f(r)$ was assigned. Next, execute FreeTail(t) and receive the a-request a of level b from tank$[b,t]$. We then find a level to which we assign the request a. Starting from level t, we see if the level contains at least one a-request, until we reach level $b+1$. Let ℓ be the first such level. Then execute AppendRich(ℓ,a), which creates tank$[b,\ell]$. If there is no such level ℓ between t and $b+1$, execute AppendRich(b,a). In this case, we may need Repair. Without considering the effect of Repair, it costs either 1 or 2. If it costs 1, the number of locally rich levels stays unchanged or decreases by 1, and if it costs 2, the number of locally rich levels decreases by 1.

Case (IV): The case that $\ell(f(r))$ belongs to tank$[b,t]$, $\ell(f(r)) = t$, and t is not locally rich. Extended-Lazy behaves exactly the same way as Case (III). In this case, we do not need Repair by a similar observation as Case (II). It costs either 1 or 2, and the number of locally rich levels increases by 1.

Case (V): The case that $\ell(f(r))$ belongs to tank$[b,t]$, $\ell(f(r)) = b$, and t is locally rich. First, release $f(r)$. If $f(r)$ was the request assigned to tank$[b,t]$, stop here; otherwise, do the following: Execute FreeTail(t) and receive the a-request a of level b from tank$[b,t]$. Then, using AppendLeft(b,a), assign a to the vertex to which $f(r)$ was assigned. In this case, we may need Repair because tank$[b,\ t]$ becomes unassigned. The incurred cost is 1 or 0, and the number of locally rich levels decreases by 1 without considering the effect of Repair.

Case (VI): The case that $\ell(f(r))$ belongs to tank$[b,t]$, $\ell(f(r)) = b$, and t is not locally rich. Extended-Lazy behaves exactly the same way as Case (V). In this case, we do not need Repair for the same reason as Case (II). The cost is 1, and the number of locally rich levels increases by 1.

Case (VII): The case that $\ell(f(r))$ belongs to tank$[b,t]$, $b < \ell(f(r)) < t$, and t is locally rich. First, release $f(r)$. If $f(r)$ was assigned to the rightmost assigned vertex at $\ell(f(r))$, do nothing. Otherwise, using FreeTail$(\ell(f(r)))$, receive an a-request a' of level $\ell(f(r))$, and using AppendLeft$(\ell(f(r)),a')$, assign a' to the

vertex to which $f(r)$ was assigned. Next, execute FreeTail(t) and receive the a-request a of level b from tank$[b, t]$. We then find a level to which we assign the request a in the same way as Case (III). Starting from level $\ell(f(r))$, we see if the level contains at least one a-request, until we reach level $b+1$. Let ℓ be the first such level. Then execute AppendRich(ℓ, a), which creates tank$[b, \ell]$. If there is no such level ℓ between $\ell(f(r))$ and $b+1$, execute AppendRich(b, a). In this case, we may need Repair. Without considering the effect of Repair, it costs either 1 or 2. If it costs 1, the number of locally rich levels is unchanged or decreases by 1, and if it costs 2, the number of locally rich levels decreases by 1.

Case (VIII): The case that $\ell(f(r))$ belongs to tank$[b, t]$, $b < \ell(f(r)) < t$, and t is not locally rich. Extended-Lazy behaves exactly the same way as Case (VII). In this case, we do not need Repair for the same reason as Case (IV). The cost is 1 or 2. The number of locally rich levels increases by 1 or 2 when the cost is 1, and by 1 when the cost is 2.

Recall that after executing Cases (I), (III), (V), or (VII), the OVSF code tree may not satisfy semi-compactness. In such a case, however, there is only one level that breaks the condition of semi-compactness, and furthermore, there is only one broken condition, namely (ii) or (v) (again, the proof is omitted). If (ii) is broken at level ℓ, level ℓ consists of, from left to right, a sequence of dead vertices up to some point, then one non-dead vertex v, and then again a sequence of (at least one) dead vertices. This non-dead vertex was called a "hole" in [2]. We also use the same terminology here, and call level ℓ a *hole-level*. If (v) is broken at level ℓ, ℓ is a bottom of a tank tank$[\ell, t]$ and is rich. Furthermore, level ℓ consists of, from the leftmost vertex, a sequence of 0 or more dead vertices, a sequence of 1 or more non-dead vertices, and then the leftmost level-ℓ descendant of tank$[\ell, t]$ (which is non-dead by definition). We call level ℓ a *rich-bottom-level*. A level is called a *critical-level* if it is a hole-level or a rich-bottom-level.

The idea of Repair is to resolve a critical-level one by one. When we remove a critical-level ℓ by Repair, it may create another critical-level. However, we can prove that there arises at most one new critical level, and its level is higher than ℓ. Hence we can obtain a semi-compact assignment by applying Repair at most h times.

Let us explain the operation Repair (again because of a space restriction, we will give only a rough idea and omit detailed descriptions). If ℓ is a hole-level and ℓ is not a bottom of a tank, then we release the a-request a assigned to the rightmost assigned vertex at level ℓ, and reassign it to the hole to fill the hole by AppendLeft(ℓ, a). If ℓ is a hole-level and ℓ is a bottom of a tank v(=tank$[\ell, t]$), then there is a vertex u that is the leftmost level-ℓ descendant of v. Recall that the a-request virtually assigned to v is actually a request for level ℓ and is assigned to u. We release this request a using FreeTail(t) and perform AppendLeft(ℓ, a). Finally, if ℓ is a rich-bottom-level, then we will do the same operation, namely, release the a-request a from the tank, and execute AppendLeft(ℓ, a).

Lemma 2. Extended-Lazy *preserves the semi-compactness on r-requests.*

Proof. Similarly as Lemma 1, we will check that five conditions (i) through (v) of semi-compactness are preserved for each application of Cases (I) through (VIII)

(followed by appropriate number of applications of Repair). Because of the space restriction, it is omitted. □

4 Competitive Analyses of Extended-Lazy

First, we estimate the cost and the increase in the number of locally rich levels incurred by applications of Repair. By a single application of Repair, the cost of 1 is incurred and the number of locally rich levels increases or decreases by 1. In case that the number of locally rich levels increases by 1, the resulting OVSF code tree is semi-compact. On the other hand, if the number of locally rich levels decreases by 1, we may need one more application of Repair. Hence, if Repair is executed k times, then the total cost of k is incurred, and the number of locally rich levels decreases by $k-2$ or k. (In the case of $k=1$, "decreases by $k-2$" means "increases by 1".)

Table 1. The costs and increases in the number of locally rich levels for each execution of Extended-Lazy.

Case	(1)	(2)	(3)	(4)	(5)		(6)
Cost	1	1	3	2	2	4	3
Increase	≤ 1	≤ 1	0	≤ 1	≤ 1	0	0

Case	(I)	(II)	(III)		(IV)	(V)	(VI)	(VII)		(VIII)	
Cost	$\leq k+1$	≤ 1	$k+1$	$k+2$	≤ 2	$\leq k+1$	1	$k+1$	$k+2$	1	2
Increase	$\leq -k+1$	1	$\leq -k+2$	$\leq -k+1$	1	$\leq -k+1$	1	$\leq -k+2$	$\leq -k+1$	≤ 2	1

Then, let us estimate the cost and the increase in the number of locally rich levels for each of the cases (1) through (6) and (I) through (VIII) of Extended-Lazy. From the observations of Sects. 3.1 and 3.2, and the above observation on Repair, these quantities can be calculated as in Table 1. There are two values in Case (5): Left and right values correspond to the cases where Cases (2) and (3), respectively, are executed after Case (5). There are also two values in Cases (III), (VII), and (VIII), which correspond to behaviors of Extended-Lazy. In the lower table, k denotes the number of applications of Repair. One can see that, from the upper table, the sum of the cost and the increase in the number of locally rich levels is at most 4 for serving an a-request. This happens when Extended-Lazy executes Case (5) followed by Case (3). Similarly, by the lower table, the sum of the cost and the increase in the number of locally rich levels for serving one r-request is at most 3, which happens in Cases (III), (IV), (VII), and (VIII).

Now, we are ready to calculate the competitive ratio of Extended-Lazy. For an arbitrary input sequence σ, let A and R be the set of a-requests and the set of r-requests in σ, respectively. It is easy to see that the cost of an optimal offline algorithm is at least $|A|$ because each a-request incurs a cost of 1 in any algorithm. We then estimate the cost of Extended-Lazy. For $a \in A$ and $r \in R$, let c_a and c_r be the costs of Extended-Lazy for serving a and r, respectively. The cost of Extended-Lazy for σ is then $\sum_{a\in A} c_a + \sum_{r\in R} c_r$. Also, for $a \in A$

and $r \in R$, let p_a and p_r be the increases in the number of locally rich levels caused by Extended-Lazy in serving a and r, respectively. Define P to be the number of locally rich levels in the OVSF code tree at the end of the input σ. Then, $P = \sum_{a \in A} p_a + \sum_{r \in R} p_r$ since there is no locally rich level at the beginning. The cost of Extended-Lazy for σ is

$$\begin{aligned}
\sum_{a \in A} c_a + \sum_{r \in R} c_r &\leq \sum_{a \in A} c_a + \sum_{r \in R} c_r + P \\
&= \sum_{a \in A} (c_a + p_a) + \sum_{r \in R} (c_r + p_r) \\
&\leq \sum_{a \in A} 4 + \sum_{r \in R} 3 \qquad (1) \\
&= 4|A| + 3|R| \\
&\leq 7|A|. \qquad (2)
\end{aligned}$$

(1) is due to the above analysis, and (2) is due to the fact that $|R| \leq |A|$ since for each r-request, there must be a preceding a-request corresponding to it. Now, the following theorem is immediate from the above inequality.

Theorem 1. *The competitive ratio of* Extended-Lazy *is at most 7.*

Next, we give a lower bound on the competitive ratio of Extended-Lazy.

Theorem 2. *The competitive ratio of* Extended-Lazy *is at least* $7 - \epsilon$ *for any positive constant* $\epsilon > 0$.

Proof. As one can see in the upper bound analysis, the most costly operations are Case (5) followed by Case (3) for a-requests and Cases (III), (IV), (VII), and (VIII) for r-requests. We first give a short sequence which leads an OVSF code tree of Extended-Lazy to some special configuration, and after that, we give a-requests and r-requests repeatedly, for which Extended-Lazy executes Case (5) followed by Case (3), or Case (VIII) almost every time. Because of the space restriction, we omit the complete proof. □

5 A Lower Bound

Theorem 3. *For any positive constant* $\epsilon > 0$*, there is no* $(2 - \epsilon)$*-competitive online algorithm for the online OVSF code assignment problem.*

Proof. Consider an OVSF code tree of height h (where h is even), namely, the number of leaves are $n = 2^h$. First, an adversary gives n a-requests of level 0 so that the vertices of level 0 are fully assigned, by which, an arbitrary online algorithm incurs the cost of n. Then, depending on the assignment of the online algorithm, the adversary requires to release one a-request from each subtree rooted at a vertex of level $h/2$. (Hereafter, we simply say "subtree" to mean a subtree of this size.) Since there are $\sqrt{n}$ such subtrees, the adversary gives $\sqrt{n}$ r-requests in total. Next, the adversary gives an a-request a_1 of level $h/2$. To assign

a_1, the online algorithm has to make one of subtrees empty by reassignments, for which the cost of at least $\sqrt{n}-1$ is required.

Again, depending on the behavior of the online algorithm, the adversary requires to release $\sqrt{n}$ a-requests of level 0 *uniformly* from each subtree except for the subtree to which a_1 is assigned. Here, "uniformly" means that the numbers of r-requests for any pair of subtrees differ by at most 1; in the current case, the adversary requires to release two a-requests from one subtree, and one a-request from each of the other $\sqrt{n}-2$ subtrees. Subsequently, the adversary gives an a-request a_2 of level $h/2$. Similarly as above, the online algorithm needs $\sqrt{n}-2$ reassignments to assign a_2.

The adversary repeats the same operation $\sqrt{n}$ rounds, where one round consists of $\sqrt{n}$ r-requests to release a-requests of level 0 uniformly from subtrees, and one a-request of level $h/2$. Eventually, all initial a-requests of level 0 are removed, and the final OVSF code tree contains $\sqrt{n}$ a-requests of level $h/2$.

The total cost of the online algorithm is $n+\sqrt{n}+(\sqrt{n}-1)+(\sqrt{n}-2)+(\sqrt{n}-2)+\cdots = n+\sqrt{n}+\sum_{i=1}^{\sqrt{n}}(\sqrt{n}-\lceil\frac{\sqrt{n}}{\sqrt{n}+1-i}\rceil) > 2n-\sqrt{n}(\log\sqrt{n}+(\sum_{i=1}^{\sqrt{n}}\frac{1}{i}-\log\sqrt{n}))$. On the other hand, the cost of an optimal offline algorithm is $n+\sqrt{n}$ since it does not need reassignment. Hence, the competitive ratio is at least

$$\frac{2n-\sqrt{n}(\log\sqrt{n}+(\sum_{i=1}^{\sqrt{n}}\frac{1}{i}-\log\sqrt{n}))}{n+\sqrt{n}} = 2-\frac{\sqrt{n}(\log\sqrt{n}+2+(\sum_{i=1}^{\sqrt{n}}\frac{1}{i}-\log\sqrt{n}))}{n+\sqrt{n}}.$$

Since $\lim_{n\to\infty}(\sum_{i=1}^{\sqrt{n}}\frac{1}{i}-\log\sqrt{n}) = \gamma$ ($\gamma \simeq 0.577$) is the Euler's constant, the term $\frac{\sqrt{n}(\log\sqrt{n}+2+(\sum_{i=1}^{\sqrt{n}}\frac{1}{i}-\log\sqrt{n}))}{n+\sqrt{n}}$ becomes arbitrarily small as n goes infinity. □

References

1. Adachi, F., Sawahashi, M., Okawa, K.: Tree-structured generation of orthogonal spreading codes with different lengths for forward link of DS-CDMA mobile radio. Electronics Letters 33(1), 27–28 (1997)
2. Chin, F. Y. L., Ting, H. F., Zhang, Y.: A constant-competitive algorithm for online OVSF code assignment. In Proc. of The 18th International Symposium on Algorithms and Computation (ISAAC), LNCS 4835, 452–463 (2007)
3. Chin, F. Y. L., Zhang, Y., Zhu, H.: Online OVSF code assignment with resource augmentation. In Proc. of The 3rd International Conference on Algorithmic Aspects in Information and Management (AAIM), LNCS 4508, 191–200 (2007)
4. Erlebach, T., Jacob, R., Mihaľák, M., Nunkesser, M., Szabó, G., Widmayer, P.: An algorithmic view on OVSF code assignment. Algorithmica 47(3), 269–298 (2007)
5. Erlebach, T., Jacob, R., Tomamichel, M.: Algorithmische aspekte von OVSF code assignment mit schwerpunkt auf offline code assignment. Student thesis as ETH Zürich
6. Forišek, M., Katreniak, B., Katreniaková, J., Královič, R., Koutný, V., Pardubská, D., Plachetka, T., Rovan, B.: Online bandwidth allocation. In Proc. of The 16th Annual European Symposium on Algorithms (ESA), LNCS 4698, 546–557 (2007)
7. Minn, T., Siu, K. Y.: Dynamic assignment of orthogonal variable-spreading-factor codes in W-CDMA. IEEE Journal on Selected Areas in Communications 18(8), 1429–1440 (2000)